NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DETECTING POTENTIAL SYNCHRONIZATION CONSTRAINT DEADLOCKS FROM FORMAL SYSTEM SPECIFICATIONS

by

Jeffrey Mark Schweiger

March, 1992

Thesis Advisor: Valdis Berzins

Approved for public release; distribution is unlimited.

ASSIFICATI[illegible]

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School / 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000	7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
DETECTING POTENTIAL SYNCHRONIZATION CONSTRAINT DEADLOCKS FROM FORMAL SYSTEM SPECIFICATIONS

12. PERSONAL AUTHOR(S)
Schweiger, Jeffrey Mark

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Master's Thesis	FROM 03/89 TO 03/92	March 1992	81

16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Deadlock, Formal Specification, Software Engineering, Distributed Systems, Concurrent Systems, Regular Expressions, Sychronization

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This thesis describes a conceptual design for a software tool for automatic detection of synchronization constraint deadlock from the formal specification of a distributed system. The formal specification language Spec is used to define the distributed system. The basic algorithm used is introduced using a graphical representation, and its operation illustrated via an example.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Valdis Berzins	22b. TELEPHONE *(Include Area Code)* (408) 646-2461 / 22c. OFFICE SYMBOL CS/Be

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Detecting Potential Synchronization Constraint Deadlocks from Formal System Specifications

by

Jeffrey Mark Schweiger
Lieutenant Commander, United States Navy
S.B., Massachusetts Institute of Technology, 1975
M.S., Naval Postgraduate School, 1982

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

Author: [illegible]
Jeffrey Mark Schweiger

Approved by: [illegible]
Valdis Berzins, Thesis Advisor

[illegible]
Luqi, Second Reader

[illegible]
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

This thesis describes a conceptual design for a software tool for automatic detection of synchronization constraint deadlock from the formal specification of a distributed system. The formal specification language Spec is used to define the distributed system. The basic algorithm used is introduced using a graphical representation, and its operation illustrated via an example.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to express my deep gratitude to my parents for their support. I also want to acknowledge the many friends, colleagues and advisors I have had during my research. It would take too long to acknowledge them all (and would resemble the departmental roster). My advisors for this Master's Thesis, Professors Valdis Berzins and Luqi, made this process an enjoyable one. The members of my doctoral committee, not involved with this thesis, Professors Man-Tak Shing, Bert Lundy, Carl Jones and Jim Eagle, have been there when I needed their expertise and support. Hopefully, it will not be too long before they get to help me finish my dissertation. Professor Tim Shimeall has also been of great help during my research. Jon Hartman, Dave Pratt and Commander Gary Hughes were valued friends as I performed this work. In addition to his role as Curricular Officer, Commander Tom Hoskins was also a vital friend and advisor.

I would be remiss if I didn't single out the administrative, technical and operations staff of the Computer Science department. The effort and friendship, on a daily basis, of Hank Hankins, Al Wong, Rob Cortilla, Hollis Berry, Bernadette Brooks, Rosalie Johnson, Walt Landaker, Chuck Lombardo, Shirley Oliveira, Frank Palazzo, Jenny Stevens, Russ Whalen, Sue Whalen, Mike Williams, and Terry Williams is what really makes this department function.

I. INTRODUCTION

A. PURPOSE

Our goal is to develop a software tool to automatically detect the possibility of deadlock in a distributed system from the formal specification of that system. The software tool will take a formal specification of a distributed system as input, and will evaluate whether or not that system, as specified, has the potential for deadlock. The tool will indicate whether or not the design has potential for deadlock, and will produce a graphic depiction of the specification. The initial scope of this tool will be limited to synchronization constraints on sequences of events that can be described using regular expressions. Accesses to such shared resources are subject to synchronization constraints that can lead to deadlock situations if not designed properly. The proposed tool will detect this type of design fault.

Formal specifications can provide a precise 'black-box' description modeling the behavior of a software system. The behavior modeled by the formal specification consists of the interactions of a software system with other software systems or the external world (e.g., operator commands).

Distributed systems are collections of computers that appear as a single machine to its users (Tanenbaum, 1984). In a distributed system the interactions modeled by the formal specification extend to the use of shared, critical resources (such as common data bases).

B. DEADLOCK

Informally, a state of deadlock occurs if a process in a system, or an entire distributed system, waits for a particular event that cannot occur (Deitel, 1990). For example, a deadlock occurs if processor A has control of resource 1 and needs resource 2 to complete a task, while processor B has control of resource 2 and needs resource 1 to completes its task and neither processor will yield the resource it controls until it completes its assigned task. Each processor is waiting for a particular event that cannot occur (the availability of an unavailable resource) and is unable to proceed. This example describes a two-process deadlock. The example illustrates a *resource* deadlock (Chandy, 1983), a deadlock situation that arises because processes are permanently waiting for resources held by each other.

A similar situation occurs when a system (or module) enters a state from which no transitions to other states are possible (i.e., a "trap state"). Such a process is not waiting for an event to occur; there are no events specified that could allow the module to return to a state from which useful work might be performed. This might be caused by error handling policies that did not return the module to an operating state (abnormal termination). This module, however, may need to generate or pass a message or event to another module or system, causing that process to block. This is an example of a *communications* deadlock. A process deadlocked because of communications may be able to proceed upon receipt of a message from one of several processes from which it may be communicating.

The major difference between resource and communications deadlocks is that, in the case of resource deadlock, processes cannot proceed until they receive all of the resources that they require. In the case of a communications deadlock, the receipt of at least one of several potential messages may permit the resumption of execution. The communications model is considered to be more general than the resource model of deadlock. (Chandy, 1983)

Deadlock and certain associated terms are more formally defined in Chapter III.

C. FORMAL SOFTWARE SPECIFICATION METHODOLOGY

Formal software specification methods use formal languages. A formal language is a set of finite length strings of symbols over some alphabet, where the alphabet is "a finite set of symbols" (Hopcroft, 1979). With respect to computer languages, the alphabet "consists of the permissible keywords, variables, and symbols of the language" (Sudkamp, 1988). In applying these concepts to the specification of software systems:

> "Formal methods used in developing computer systems are mathematically based techniques for describing system properties. Such formal methods provide frameworks within which people can specify, develop, and verify systems in a systematic, rather than ad hoc, manner.
>
> A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides the means of precisely defining notions like consistency and completeness and, more relevantly, specification, implementation, and correctness. It provides the means of proving that a specification is realizable, proving that a system has been implemented correctly, and proving properties of a system without necessarily running it to determine its behavior." (Wing, 1990)

We can use a formal language consisting of a "clearly defined syntax and semantics" to formally specify and describe the interfaces of a system or component (Berzins, 1988).

There are many formal specification methodologies available. Some are frequently found in a graphical form, such as Petri Nets (Murata, 1989), Communicating Finite State Machines (CFSM) (Brand, 1983), and Systems of Communicating Machines (Lundy, 1988). Others are normally found in a textual form, but can be converted into a graph form for deadlock detection analysis. For this project, one of the latter, the formal specification language Spec (Berzins and Luqi, 1991), (Berzins, 1991), is used. Spec, in particular, concisely describes the atomic transactions that can cause the occurrence of deadlock.

The basic units or modules of Spec are definitions, functions, types, machines, and instances. A machine is a system with a memory capable of remembering a state. A type is a module that defines an abstract data type, consisting of a set of values and a set of primitive operations on that value set. If a type module has operations that modify the value set or change existing instances of the type, it has an internal state.

Chapter II contains a more detailed discussion of formal specification methodologies.

D. SIMPLIFYING ASSUMPTIONS

Some of the approaches previously used in evaluating deadlock potential have been shown to be undecidable in the general case. It has not yet been demonstrated whether or not this is true for the approaches I am investigating. The initial algorithm development discussed in this thesis will be restricted to situations that can be proved decidable. To ensure this, I will restrict my evaluation to those systems whose synchronization constraints can be represented using regular expressions.

E. OVERVIEW

In this chapter, I have introduced the goals of this research and informally discussed the topics of deadlock and formal software specification methodologies. In the remainder of this thesis, these topics are explained in more detail, and a method for automatic detection for deadlock detection is presented.

Chapter II summarizes the background for my research, and includes a review of several formal software specification methodologies. I also describe some previous approaches to detecting deadlock potential in distributed system, and the applicability of existing software tools in this endeavor.

In Chapter III, I formally define deadlock and certain related terms, and summarize the requirements analysis for a deadlock detection software tool. I also discuss my rationale for choosing Spec as the specific formal specification methodology used in the model. This chapter also outlines the algorithmic approach I propose for automatic detection of deadlock potential, and gives an example of its use.

Chapter IV summarizes the results of the research and describes proposed extensions and additional necessary work.

II. BACKGROUND

A. REVIEW OF FORMAL SPECIFICATION METHODOLOGIES

There are many different methodologies available for use in the formal specification of systems. These methodologies may be categorized in many different ways. These characteristics include, but are not limited to, whether or not a method is graphical in nature, is model based or property based, what the underlying mathematical system is, intended for sequential and/or concurrent systems, etc. The approach I take toward deadlock detection in Chapter III is derived from graph theory, but requires a character oriented machine readable specification method. In addition, to be useful for deadlock detection, the specification methods used must be able to specify characteristics of concurrent, distributed systems. Separate, sequential systems that do not interact with other systems are of no interest for this application. Formal methods that can be used for specifying large software systems are of particular interest, as this is a requirement for real-world applications.

In the following paragraphs, I summarize several such methodologies, commenting on many of their characteristics. While the list is clearly not all inclusive, it is representative of the many specification formalisms available.

1. Petri Nets

A Petri net is a graphical and mathematical tool useful in expressing concurrency and parallelism.

Petri nets consist of a particular type of directed graph, known as a bipartite graph, and an initial state, or *initial marking*. The directed, bipartite graph of a Petri net may be weighted, and has two kinds of nodes, *places* and *transitions*. Arcs in a Petri net start at a place and terminate at a transition, or start at a transition and end at a place. States are represented by Petri net *markings*, where a marking is a function that assigns a non-negative integer value to each place. This assignment is represented by *tokens*, where if the non-negative integer value, k, is assigned to place, **p**, we say that **p** has **k** tokens. The formal definition of a Petri net is given in Table I (Murata, 1989).

TABLE I FORMAL DEFINITION OF A PETRI NET

A Petri net is a 5-tuple, $PN=(P,T,F,W,M_0)$ where:

$P=\{p_1,p_2,...,p_m\}$ *is a finite set of places,*
$T=\{t_1,t_2,...,t_n\}$ *is a finite set of transitions,*
$F\subseteq(P\times T)\cup(T\times P)$ *is a set of arcs (flow relation),*
$W:F\rightarrow\{1,2,3,...\}$ *is a weight function,*
$M_0:P\rightarrow\{0,1,2,3,...\}$ *is the initial marking,*
$P\cap T=\emptyset$ *and* $P\cup T\neq\emptyset$.

A Petri net structure $N=(P,T,F,W)$ without any specific initial marking is denoted by N.

A Petri net with the given initial marking is denoted by (N,M_0).

The conventional graphical representation of Petri nets use circles for places and bars as transitions. Tokens are represented by small dots. A simple Petri net is shown in

Figure 2.1. Petri net weights may be represented by either multiple arcs of weight one between places and transitions, or by labeling arcs with a specific weight.

In modeling an event or computational step, the transition represents the event, and places, conditions. The

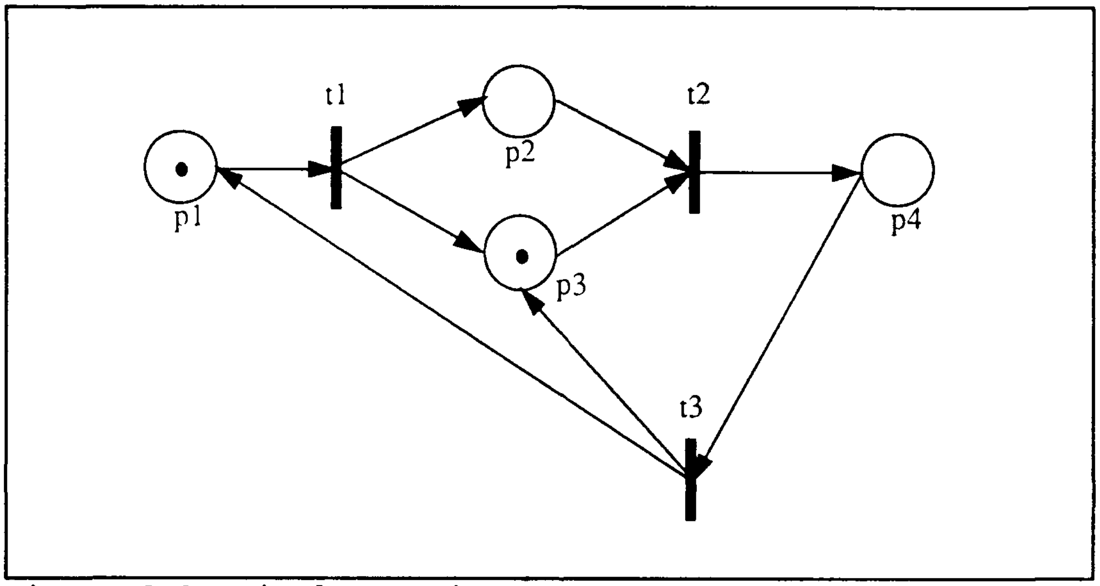

Figure 2.1 Simple Petri Net

places prior to the transition, or *input places*, represent *pre-conditions* and the *output places* following the transition, *post-conditions*.

Events occur, and the marking or state changes, when a Petri net transition *fires*. A transition, *t*, fires after it has been *enabled*, which means that each input place, p_i, for that transition is marked with at least the number of tokens indicated by the weight of the arc from p_i to t. In a Petri net where all arcs have weight one, the firing of a transition removes one token from each input place of the transition and adds one token to each output place of the transition.

In classical Petri nets, the time from when a transition, t, is enabled until it fires is indeterminate. Similarly, if two different transitions, t_1, t_2, are enabled concurrently, the order of firing of t_1 and t_2 is indeterminate (Coolahan, 1983). Transitions can therefore be *in conflict* if the firing of one enabled transition causes the *disabling* of another enabled transition.

Petri nets can be used in the modeling of finite state machines, parallel activities, dataflow computation, communication protocols, multiprocessor systems, synchronization control of multiprocessor and distributed-processing systems and formal languages. Behavioral properties including reachability, boundedness, liveness, reversibility and fairness have been modeled using Petri nets (Murata, 1989).

Reachability in a Petri net is the determination of whether, given an initial marking, M_0, and a desired marking, M_n, there exists a sequence of firings that will transform M_0 to M_n. (Kosaraju, 1982) and (Mayr, 1984) have shown that Petri net reachability is decidable, though it does require at least exponential space and time. Reachability is the fundamental, underlying method used in evaluating all of the Petri behavioral properties noted above.

A Petri net *(N, M_0)* is called *k-bounded* (or *bounded*) if, for any marking reachable from M_0, the number of tokens in each of the places of the Petri net never exceeds a finite number *k*. If places in Petri nets are used to represent buffers, the boundedness property can be used to guarantee the lack of the buffers overflowing.

A Petri net *(N, M_0)* is considered to be *live* if it is possible to eventually fire any transition of the net from the current marking, by progressing through some further firing sequence. Thus, a system described by a live Petri net is guaranteed to be free of deadlock, no matter what firing sequence is chosen. (Murata, 1989)

Reversibility is a property where the initial marking M_0 of a Petri net is reachable from any marking reachable from M_0.

There are multiple definitions of fairness in Petri nets. Two of these are *bounded-fairness* (*B-fair*) and *global fairness*. A Petri net is *B-fair* if, for every pair of transitions in the net, there is an upper limit (or bound) on the number of times that either can fire before the other transition can fire. A Petri net is *globally* (or *unconditionally*) fair if for every firing sequence reachable from M_0, is finite, or every transition in the net appears infinitely often in the firing sequence.

Reachability and liveness analysis using Petri nets are further discussed in section B of this chapter.

2. Communicating Finite State Machines (CFSM)

The Communicating Finite State Machine (CFSM) methodology was developed for use in modelling communications protocols. The CFSM model represents and specifies communicating processes as finite state machines. Each pair of these processes are connected by a full duplex First-in First-out (FIFO) channel (that can be represented by two one-way FIFO queues) (Brand, 1983).

CFSMs are directed labeled graphs with two types of edges, *sending* and *receiving*. The edges are labeled with the name of a message. The label is prepended with a minus sign for a sending arc, and a plus sign for a receiving arc. The nodes of a CFSM represent its states, and each CFSM has its starting state identified by an *initial node*. If a CFSM node has only sending edges, it is referred to as a *sending node*. Similarly, a node with only receiving edges is a *receiving node*. A *mixed node* is one with both sending and receiving edges (Gouda, 1986), (Lundy, 1988). A formal CFSM model for an arbitrary number of processes or machines is given in Table II (Brand, 1983).

To illustrate CFSM, we will simplify to using a two machine network. Let M and N be two CFSMs sharing the same set G of messages. We will call $W = (M, N)$ a network. The global state of network W is represented by the four-tuple (m, c_m, c_n, n), where m and n are nodes (states) of M and N, and c_m and c_n represent messages from G being passed from M to

TABLE II CFSM PROTOCOL DEFINITION

A CFSM protocol is a 4-tuple,

$$(\langle S_i \rangle_{i=1}^{N}, \langle o_i \rangle_{i=1}^{N}, \langle M_{ij} \rangle_{i,j=1}^{N}, succ) \text{ where:}$$

N is a positive integer (representing the number of processes),
$\langle S_i \rangle_{i=1}^{N}$ are N disjoint sets (S_i represents the set of states of process i),
each o_i is an element of S_i (representing the initial state of process i),
$\langle M_{ij} \rangle_{i,j=1}^{N}$ are N^2 disjoint finite sets with M_{ii} empty for all i (M_{ij} represents the messages that can be sent process i to process j)
succ is a partial function mapping for each i and j),

$$S_i \times M_{ij} \rightarrow S_i \quad \text{and} \quad S_i \times M_{ji} \rightarrow S_i$$

($succ(s,x)$ is the state entered after a process transmits or receives messagex in states. It is a transmission if x is from M_{ij}, a reception if x is from M_{ji}.)

N, and *N* to *M* respectively. The initial global state of *W* is (m_0, E, E, n_0), where m_0 and n_0 are the initial states of *M* and *N*, and *E* is the empty string (Gouda, 1986, Lundy, 1988).

Figure 2.2 depicts an example two machine CFSM, representing the simple two process (user and server) protocol described in (Brand, 1983). The left-hand machine represents the user process with state 0 representing a *ready* state, state 1 a *wait* state, and state 2 a *register* state. The right-hand machine is the server process, and states 0, 1, and 2 represent the *idle*, *service*, and *fault* states, respectively.

The most common method of analysis used with CFSM is reachability analysis, where all possible global states are generated by evaluating all possible transitions in each machine. This analysis may not terminate if the FIFO queues connecting the machines are unbounded (Lundy, 1988). Even in the bounded case, the number of global states necessary to

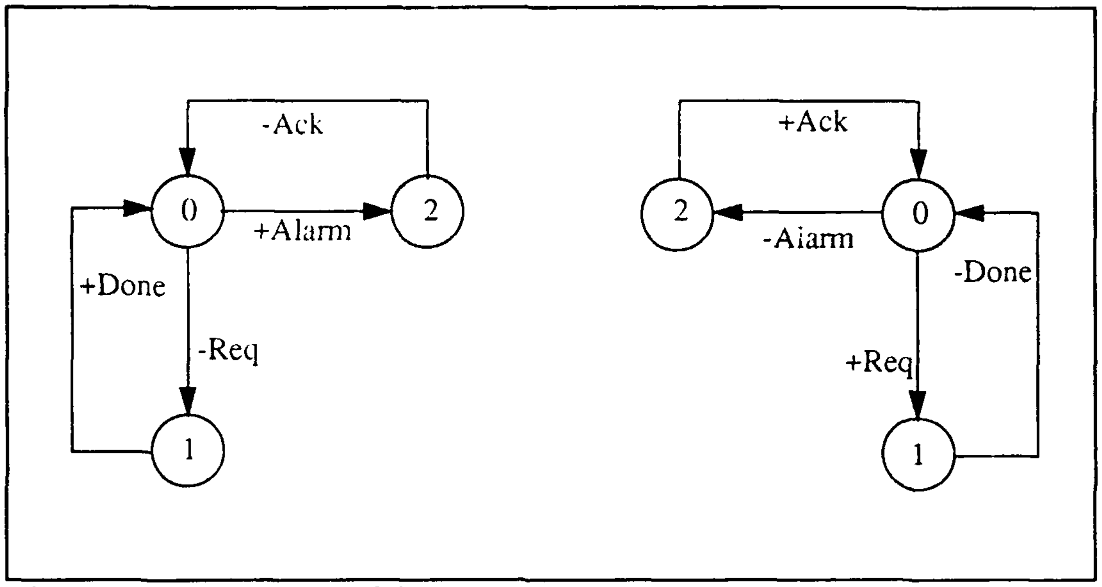

Figure 2.2 Two Machine CFSM

model a network may grow so quickly as to be impractical to calculate.

3. Systems of Communicating Machines (SCM)

Systems of Communicating Machines (SCM) is a formal description technique consisting of finite state machines and variables and is an extension of CFSM. A formal definition of the SCM model is given in Table III (Lundy, 1988). The finite state machines model entities, which are processes and channels in a protocol system, or concurrent processes in a parallel programming system. Communications between the machines is accomplished through the use of *shared variables* instead of the implicit queues used in CFSM. *Local variables* are used in the machines for various purposes, including serving as counters or for storing data blocks. Transitions between states have associated with them *enabling predicates*

TABLE III SCM MODEL

A System of Communicating Machines is an ordered pair, $C=(C,M)$, where

$$M=\{m_1, m_2, \ldots, m_n\}$$

is a finite set of machines, and

$$V=\{v_1, v_2, \ldots, v_k\}$$

is a finite set of shared variables, with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of read access variables for machine m_i, and the subset W_i the set of write access variables for m_i.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s, L_i, N_i, \tau_i)$, where

(1) S_i is a finite set of states;

(2) $s \in S_i$ is a designated state called the initial state of m_i;

(3) L_i is a finite set of local variables;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p, a), where p is a predicate on the variables of $L_i \cup R_i$, and 'a' is a partial function

$$a:\ L_i \times R_i \rightarrow L_i \times W_i$$

from the local variables and read access variables to the local variables and write access variables.

(5) τ_i: $S_i \times N_i \rightarrow S_I$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

that determine when a transition may occur and *actions* that alter variable values. The major analysis method used in SCM is known as *system state analysis*, and is analogous to the reachability analysis used in Petri Nets and CFSM. It has been shown that any protocol that can be described by CFSM can also be modeled using SCM. Under some conditions, the SCM model system state analysis will be of significantly reduced complexity compared to CFSM reachability analysis for the equivalent model. (Lundy, 1988)

4. Statecharts

Statecharts is a graphical language that models the behavior of a system by extending finite state machines (FSM's) and state-transition diagrams. As opposed to single level FSM's, statecharts are hierarchically decomposed into substates in AND/OR fashion, and assuming instantaneous broadcast communications. Single level sets of communicating FSM's have the potential for an unacceptably large number of states to be considered during analysis. The statechart approach of grouping substates into states reduces the analysis necessary on any one particular level, and makes it possible to describe independent and concurrent state components. Transitions in a statechart are labeled with optional expressions that represent the event trigger the transition, guard conditions that must be true when the triggering event occurs, and actions that are carried out if and precisely when the transition is taken. (Harel, 1990)

5. Z

The Z language is a non-graphical, abstract data type based formal specification notation. In particular, Z is based upon set theory and first-order predicate logic. Z decomposes a specification into modules known as *schemas*. These schemas describe the static (states that can be occupied, and invariant relationships maintained as the system changes states) and dynamic (possible operations, relationship between inputs and outputs, state changes possible) aspects of the system. Z is primarily considered to be a *model-oriented,* as opposed to a *property-oriented,* specification method. Z is used to define an abstract model of the system being specified. Z specifications may also describe certain aspects of systems, using axioms to denote certain conditions, or properties, that must be satisfied by the system. In this way Z is also property-oriented. (Spivey, 1988), (Spivey, 1989)

Z is a sequentially based method, and has no built-in concurrency features. When using Z for specifying concurrent systems, it must be combined with a method that supports concurrency, such as CSP. Because of this limitation, Z is not suitable for deadlock detection analysis.

6. VDM

VDM (Vienna Development Method) meta-language is similar to Z in that it is a non-graphical notation based upon abstract data types and predicate logic. (Spivey, 1988), (Cohen, 1989), (Jones, 1990)

VDM is model oriented and makes use of several data types (including sets, lists and maps) in constructing specifications. Like Z, VDM is state-based and designed for specifying sequential, as opposed to concurrent systems (Wing, 1990), (Woodcock, 1990), (Berzins, 1991). This lack of built-in support for specification and modeling of concurrent systems limits the utility of using VDM in detecting potential deadlocks.

7. Larch

Larch is a property-oriented method that uses both axiomatic and algebraic specifications (Wing, 1990). Larch is actually a family of languages, consisting of the Larch Shared Language and a series of Larch interface languages, specific to particular programming languages. The Larch Shared Language describes algebraic specifications with equations defining relations between operators. Operators and their properties are specified by use of *traits*. Traits initially introduce Larch operators, and specify their domains and ranges. The specification then constrains the operators using equations to define the relations between them. These equations are known as *terms*. A trait's *theory* is the set of theorems that can be proven about its terms. (Guttag, 1985) This allows the development of automated tools for verifying Larch traits (Wing, 1990).

Larch was developed to specify sequential (non-concurrent) programs, and explicitly does not include the ability to specify atomic actions (Guttag, 1985). Larch is therefore unsuitable for use in deadlock potential analysis.

8. SDYMOL

SDYMOL is a design language intended for use in describing communications and synchronization between individual sequential processes in concurrent systems. It is a simplified version of the Dynamic Modeling Language (DYMOL). SDYMOL specifies process interaction, but only abstractly describes the internal requirements for the individual sequential processes. SDYMOL is model-oriented in how it initially defines a process, and property-oriented in SDYMOL expression evaluation of specifications.

SDYMOL processes use memory locations called *buffers* to hold messages that are sent or received via *ports*. To send or receive messages, the ports must be connected by a *channel* to a communications *link*. Ports are classified as either *inbound* or *outbound*. Inbound ports may be connected to several links, but outbound ports are limited to a single link. The links, themselves, are unbounded, unordered repositories for messages sent from outbound ports, but not yet received via inbound ports.

SDYMOL syntax is based on Algol 60 and provides a standard set of control flow constraints. Decisions based on internal process computations are represented as non-deterministic choices. Semantically null user-defined identifiers are used to represent the internal process computations. These serve as placeholders for future system elaboration.

The behavioral properties of concurrent software systems described using SDYMOL can be characterized by use of constrained expression analysis. (Dillon, 1988)

9. Communicating Sequential Processes (CSP)

Like SDYMOL, CSP represents concurrent systems by means of communications over channels between sequential processes. CSP is both model-oriented (for initial process specification) and property-oriented (for stating and proving properties about the specification) (Wing, 1990). Hoare describes the concept of a process as a mathematical abstraction of the interactions between a system and its environment. *Traces* are used to record the event sequences of a process. The events are considered as actions without duration, and may require concurrent participation by more than one process. Processes can be specified in advance of actual implementation by describing the properties of its traces. One particular class of events is known as *communication*. (Hoare, 1985)

Communications are events described by a pair or tuple comprised of a *channel* and a *message*. CSP uses the convention that processes are unidirectional (input or output) and exist between only two processes. Communication is synchronized, the transmitting process transmits simultaneously with reception by the receiving process. If buffering is necessary, a separate buffer process must be specified.

The essential properties of CSP operators are described by algebraic laws. These, coupled with proof rules, permit the analysis of CSP specifications by stating and proving properties about its traces.

CSP, like SDYMOL, can be used to analyze the possible behavior of concurrent systems by use of constrained expressions (Dillon, 1988). The formalism, itself, is designed specifically to model concurrency, and should therefore be of use in the analysis of potential deadlock.

10. Calculus of Communicating Systems

Like CSP, Milner's Calculus of Communicating Systems (CCS) is a process algebra based method for specification and verification of concurrent systems. CCS models system behavior with trees of system states and events. The individual events are indivisible and are also known as *atomic actions* and represent the transitions between system states. The actions are the basic entity of CCS systems. Concurrent systems are composed of independent agents, and a synchronized communication between two such agents is considered as a single action. The transitions in CCS trees are labelled to show how they are synchronized, and are referred to as *Synchronization Trees (ST)*.

CCS was developed explicitly to reason about concurrent systems (Woodcock, 1990). However, the processes modeled in CCS are represented as though performing only one action at a time, interleaving the actions in an arbitrary manner. There is some argument that this reduces the ability to properly model parallelism. The analysis of some behavioral properties of concurrent systems may be negatively effected by this representation. In the interleaving approach, actions must be assumed to be atomic entities. As such, CCS may be useful in the evaluation of potential deadlock. (Milner, 1980), (Wing, 1990), (Goltz, 1990).

11. Temporal Logic

Temporal logic is a property-oriented specification method intended for use with concurrent and distributed systems (Wing, 1990). It is an extension of traditional logic, as represented by Boolean algebra and predicate calculus, in that it introduces additional operators to deal with temporal sequences (Bochmann, 1982). Traditional predicate logic expressions can be assumed to specify system properties at some arbitrary given time, assumed here to represent the current or present time. The operators used in temporal logic have not been standardized, but many are commonly used. These include a *henceforth* operator ($\Box$), an *eventually* operator (represented both as $\Diamond$ or $\triangledown$), a *next* operator (sometimes seen as $\bigcirc$) and a *while* operator. Henceforth is used to indicate that the predicate operated upon will hold in all future states (for example the expression $A \Rightarrow \Box P$ indicates that whenever A is true in the present time, predicate P is true and will remain true forever). The eventually operator indicates that the predicate will eventually be true at some future time, possibly the present, but not necessarily remain true. The next operator is used to signify that the predicate will hold in the *next* state (or instance of time) to be considered. The expression *A* **while** *P* indicates that *A* is true only while *P* is true. Temporal logic specifications are unstructured sets of predicates. Each predicate expression represents a particular property that must be satisfied by the system implementation.

Temporal logic may be used to describe the semantics of concurrent programs, and specify the conditions necessary for atomic actions to be performed (Pneuli, 1979). Temporal logic could be used to evaluate behavioral properties of distributed and concurrent systems.

12. Transition Axioms

The use of transition axioms is also a property-oriented method of system specification. The behavior of individual system operations is described using an axiomatic method. A state model is used to highlight the sequential aspects of individual operations.

The transition axiom method borrows from other specification formalisms to describe both sequential and concurrent features. Issues of interest in deadlock evaluation, such as synchronization conditions, and liveness and safety constraints are specified using a form of temporal logic. (Wing, 1990)

13. LOTOS

LOTOS (Language Of Temporal Ordering Specification) is a property-oriented method used for specification of communications protocols and distributed systems in general. LOTOS specifies systems by expressing the temporal relations of their *events*. These events are considered to be instantaneous atomic actions performed by the system. LOTOS systems are specified as *processes* (potentially composed of one or more subprocesses), with interfaces to the exterior environment represented by *gates.* The interactions between and among processes take place at their shared gates. LOTOS specifications are composed of a control part, based on CCS, used for describing process behavior, and an abstract data type based data part for expressing data values exchanged between processes. (Valenzano, 1990)

The events specified in LOTOS are atomic actions whose behavior needs to be characterized in the analysis of deadlock potential, and LOTOS specifications should be able to support this type of evaluation.

14. Gypsy

Gypsy is both a formal specification language, and a verifiable, high level programming language. It is descended from Pascal and was designed primarily for use in the development of methodologies for constructing verified programs and design of highly reliable communications processing, distributed and real-time software systems. The methodology developed includes the Gypsy language and the Gypsy Verification Environment for the formal verification of Gypsy programs. The Gypsy specification component is based on first order predicate calculus and the ability to define recursive functions. Program specification can take of CSP style, or property-oriented (both algebraic and axiomatic) forms. The Gypsy language contains extensive features for data abstraction, condition handling and concurrency. (Good, 1988), (Ambler, 1977), (Carranza, 1989)

Gypsy routines are specified using both *external* (visible to the external environment) and *internal* (not visible externally) specifications. External specifications consist of two parts, an *interface* specification (consisting of the name and formal parameter list) and a *functional* specification. The functional specification describes the effects that a Gypsy routine is supposed to have on its parameters and constraints in the routine's application. An example Gypsy specification is shown in Figure 2.3. (Good, 1988)

Gypsy supports the specification of concurrent systems, through the use of processes that communicate via message buffers. These buffers are finite length queues that permit only two operations, *send* (enqueue) and *receive* (dequeue). Gypsy also makes user of a limited guard condition, an *await* statement that allows the simultaneous

```
scope SORT_SPECS =
 begin

   type ELEMENT_TYPE = pending;
   type ELEMENT_SEQ = sequence of ELEMENT_TYPE;

   function ELEM_LE (E1, E2 : ELEMENT_TYPE) : BOOLEAN =
   begin
      pending
   end;

   lemma ELEM_LE_REFLEXIVE (E : ELEMENT_TYPE) = ELEM_LE (E, E);

   lemma ELEM_LE_TOTAL (E1, E2 : ELEMENT_TYPE) =
      not ELEM_LE (E1, E2) iff ELEM_LE (E2, E1);

   lemma ELEM_LE_TRANSITIVE (E1, E2, E3 : ELEMENT_TYPE) =
      ELEM_LE (E1, E2) & ELEM_LE (E2, E3) -> ELEM_LE (E1, E3);

   function SORT_SORTS (INSEQ, OUTSEQ : ELEMENT_SEQ) : BOOLEAN =
   begin
      exit (assume     RESULT
                 iff   PERMUTATION (OUTSEQ, INSEQ)
                      & SORTED_ASCENDING (OUTSEQ));
   end;

   function SORTED_ASCENDING (OUTSEQ : ELEMENT_SEQ) : BOOLEAN =
   begin
      exit (assume     RESULT
                 iff    SIZE (OUTSEQ) le 1
                      or   ELEM_LE (LAST (NONLAST (OUTSEQ)), LAST (OUTSEQ))
                          & SORTED_ASCENDING (NONLAST (OUTSEQ)));
   end;

   function PERMUTATION (OUTSEQ, INSEQ : ELEMENT_SEQ) : BOOLEAN =
   begin
      exit (assume     RESULT
             iff if OUTSEQ = NULL (ELEMENT_SEQ)
                 then INSEQ = NULL (ELEMENT_SEQ)
                 else   LAST (OUTSEQ) in INSEQ
                      & PERMUTATION (NONLAST (OUTSEQ),
                          REMOVE_ELEMENT (LAST (OUTSEQ), INSEQ))
              fi);
   end;

   lemma PERMUTATION_REFLEXIVE (S : ELEMENT_SEQ) = PERMUTATION (S, S);

   function REMOVE_ELEMENT (ELEM : ELEMENT_TYPE;
                            S : ELEMENT_SEQ) : ELEMENT_SEQ =
   begin
      exit (assume   RESULT
                 = if S = NULL (ELEMENT_SEQ)
                      then NULL (ELEMENT_SEQ)
                   elif ELEM = LAST (S)
                      then NONLAST (S)
                      else REMOVE_ELEMENT (ELEM, NONLAST (S))
                   fi);
   end;
 end;
```

Figure 2.3 Example Gypsy Specification

waiting on the completion of any one of multiple buffer operations (Ambler, 1977). Within its definition of concurrency, it should be possible to evaluate the potential for deadlock in the Gypsy specifications of concurrent systems.

15. Anna

Anna (ANNotated Ada) is an abstract data type based method. It is an extension of, and a specification language for Ada. Anna includes Ada extensions for three purposes: generalization of explanatory constructs already existent in Ada; new, mostly declarative, constructs dealing with exceptions, context clauses, and subprograms, and extensions specifically for program specification. These latter specification constructs are based on studies of program specifications, and are used mainly to specify package semantics and composite and access type use.

The Anna language definition is devoted, in large part, to defining a an implementable transformation of specifications into run-time checks. The language definition provides axiomatic semantics, to be used to verify Ada programs by a mathematical proof of consistency between the formal Anna specification and the Ada text.

Anna specifications are Ada programs with the addition of formal Anna comments. These formal comments are syntactically structured as extensions of normal Ada comments, and as such, are acceptable by standard Ada compilers. The Anna comments are divided into two categories, virtual Ada text, and annotations.

Virtual Ada text is used to define programming concepts, but can also be used to compute values not computed by the actual program, but are useful in explaining what the program does. Virtual Ada text complies with Ada lexical, syntactic, and semantic rules, with a few Anna specific exceptions. Virtual text is also not permitted to influence the computation of the underlying program, and virtual declarations may not hide entities in the underlying Ada program text.

Anna annotations are based on Boolean-valued expressions and reserved words indicating the meaning of the annotation. Different kinds of Anna annotations are associated with specific Ada language constructs. Anna includes predefined annotation concepts and operators such as the membership test, **is in**, and the *collection* attribute. Anna also extends Ada with the addition of the *quantified expressions*, **for all** and **exist**. Anna annotations contain both *logical variables* (declared in and used in the annotations), and visible Ada text *program variables*.

Anna was designed with the intention of using it with software tools. (Luckham, 1985), (Luckham, 1990)

Anna does not specifically include support for tasking and specification of concurrent computation. As such, the method has no features to support the analysis of deadlock potential. Analysis methods, such as those used for static analysis of Ada code for deadlock, should be extendable to evaluate Anna specifications.

16. Spec

Spec is a formal specification language based on predicate logic used for writing *black-box* specifications for software systems. Spec is a model-based method and is used to describe the behavior of a module at its interface. Spec uses the event model of computation (as opposed to a state model) and includes features specifically designed for use in specifying concurrent, distributed, and time critical systems. Spec *events* are used to define timing constraints. The specification of distributed systems is aided through the use of *localized states* and *atomic transactions.* Atomic transactions specify non-interference and interaction requirements between potentially concurrent processes in a distributed environment. *Defined concepts,* and inheritance mechanisms are features used for specifying large conventional systems. In describing a system Spec specifies the behavior of three types of *modules: functions, state machines, and abstract data types.* Spec *messages* are used to communicate between modules and to model event generators and iterators.

Another feature of Spec is that it is designed to support automated processing and the use of computer-aided software engineering (CASE) tools. Such tools include consistency checkers, pretty printers syntax directed editors and a Spec to Ada translator.

Spec has several features that are supportive of an analysis of deadlock potential. Of particular use are the ability to specify concurrent interactions, and time dependent constraints. The intent of Spec as a language for the design of large-scale software systems is also helpful from a practical standpoint, in that Spec is intended for use by system designers. Spec is discussed in greater detail in the next chapter. (Berzins, 1991)

B. APPROACHES TO DEADLOCK DETECTION

Some of the methods listed above have been used in analysis of distributed systems for deaαlock potential. Some of the approaches used in conjunction with those specification methods are summarized below.

1. Static Analysis

A general definition of static analysis is simply "analysis based on examining the source code or structure" of a program (Beizer, 1990). The counterpart to static analysis is *dynamic analysis*, or analysis that is done as a program executes. In a sense, for any deadlock detection approach to be of any use in program specification, it must be a static analysis method. This includes a method known as *symbolic execution*.

2. Petri Nets

The basic method used in analyzing systems specified by Petri Nets is reachability analysis. Peterson (Peterson, 1977) defines the *reachability problem* as:

"Given a marked Petri net (with marking μ) and a marking μ', is μ'reachable from μ?"

The liveness problem (are all transitions in a Petri net live (potentially fireable in all reachable markings) ?) has been shown to be equivalent to the reachability problem. Deadlock detection in Petri nets is typically based on liveness.

Reachability analysis in Petri nets makes use of *set reachability,* determining whether a set of markings is a subset of the reachability set, *R(M),* of a Petri net. *R(M)* is defined for a marked Petri net $M = (P, T, F, W, \mu)$ as the set of all markings reachable from μ. Petri net reachability analysis represents R(M) in a finite form through the use of a *reachability tree,* whose nodes represent Petri net markings, and arcs represent possible state changes resulting from transition firings. As R(M) can often be infinite, many markings are mapped into the same node of a reachability tree. A set of states are collapsed into a single node by ignoring the number of tokens when this number becomes larger than an arbitrarily chosen limit. This arbitrarily large number can be thought of as representing infinity. (Peterson, 1977)

Cooke (Cooke, 1990), defines an approach where a class of resource deadlock prone Petri nets in is first defined in predicate calculus, and then from that definition the characteristics of this class are determined. These characteristics are then used in the specification for a Prolog program to detect similar deadlocks in parallel processes represented as Petri nets. This approach is applicable only to deadlock detection for this class of Petri net.

Another approach to deadlock detection using Petri nets is suggested by (Murata, Shenker, and Shatz, 1989). This approach starts with a given Ada tasking program, and translates the program into a Petri net representation (called an Ada net). The Ada net models the communications and control flow of the original Ada program and is analyzed using structural analysis and dynamic analysis (reachability graph). The reachability graph involved is reduced in complexity from what would be anticipated of a full Ada net through the use of Petri net place and transition invariants.

3. Temporal Logic

Owicki and Lamport (Owicki, 1982) categorize the absence of deadlock as a *safety property*, that is a property that states that the program being analyzed does not enter into an undesirable state. This is considered to be a prerequisite for evaluating *liveness properties* (something good eventually happens, where the program eventually enters a desirable state). Safety properties have the general form $P \supset \Box Q$, where P and Q are immediate assertions. The assertion of the safety property general form means that if P is true upon program initiation, then Q is always true throughout program execution. Proof of $P \supset \Box Q$ requires finding an invariant assertion I ($I \supset \Box I$ is true) that requires reasoning about the program being analyzed, such that $P \supset I$ and $I \supset Q$ (these two assertions to be proved using ordinary logic).

The method proposed in (Owicki, 1982) is specific to each program analyzed, that is a separate formal system is defined for each program. Another issue is that this method requires formal modeling of complete, existing programs, and is not appropriate for analysis of incomplete program specifications.

4. Control Flow Analysis

I group into a category of *control flow analysis*, graph algorithmic methods, other than Petri nets, for tracing the control and synchronization flow of a concurrent program. A significant amount of work has been done in this regard with respect to deadlock detection algorithms specific for use with the Ada programming language. Precise static detection of deadlock in Ada programs was shown by Taylor to be intractable (cannot be done in polynomial time) (Masticola, 1991).

Taylor (Taylor, 1983) developed an algorithm for analyzing concurrent Ada programs in time exponential to the number of tasks present in the system. The method, however, is not specific to Ada and can be used with other languages that employ rendezvous synchronization (such as CSP). This algorithm generates the concurrency history for the program and determines all possible rendezvous states, infinite waits, and parallel actions.

Another approach is suggested by Masticola and Ryder (Masticola, 1991). Here, a safe polynomial time approximation algorithm is used to evaluate task synchronization in a subset of Ada (Some of the features not included were synchronization from within procedures, and possible multiple-entry loops induced by goto statements). The algorithm creates a graph model called a *sync hypergraph* and runs in time polynomial in the size of this graph. The algorithm is *safe* in that it will always detect a deadlock if one is present, but it may also result in false reports of potential deadlocks. The analysis is based on detecting possible cycles in a static representation of the control flow of the program, rejecting those cycles that will not lead to deadlock. The algorithm uses reachability analysis to detect the presence of deadlock cycles.

5. Failures Model

The failures model of CSP has been used to describe behavior properties of certain classes of static networks of communicating processes. In particular, the potential for deadlock has been evaluated. The method uses hierarchical decomposition of networks, and requires that deadlock-freedom of subnetworks and individual nodes be established, prior to being able to prove deadlock freedom in a large network. (Brookes, 1989)

6. Symbolic Execution

Symbolic execution is a static analysis method that replaces actual program execution with symbolic operations (symbolic expressions that replace the results of executing actual program expressions).

(Dillon, 1990) describes an isolation approach to verifying specific safety properties of Ada tasking programs using symbolic execution. What is meant by *isolation* is that individual Ada tasks are symbolically executed and verified independently, and then checked for interaction with other Ada tasks. This reduces the need to deal with the concurrency simulation problem of interleaving (noted in the discussion of CCS). The safety properties evaluated in this algorithm are mutual exclusion, freedom from deadlock, and absence of communication failure.

C. APPLICABILITY OF EXISTING TOOLS FOR DEADLOCK DETECTION

Many tools have been developed for use with some of the specification methods noted above. Many of these tools do not include features that support the analysis of concurrency, even if the supported specification method does (Anna is in this category).

A Petri net analyzer, known as P-NUT (Petri Net UTilities), is a research tool from UC Irvine (Morgan, 1987). Its analysis is based on general reachability graphs, and is not guaranteed to terminate (the reachability graphs used may be infinite).

The symbolic execution research discussed in (Dillon, 1990) suggests an approach for the development of automated *symbolic execution* tools for concurrent programs. However, no such tool is yet generally available.

(Masticola, 1991) describes an implemented flowgraph research tool for detecting deadlock in a subset of Ada. This tool, however, requires the use of already written source code, and not an initial program specification.

In summary, all tools available for the detection of deadlock do not support the detection of deadlock potential from the formal specification of a distributed system.

III. DEVELOPMENT OF THE DEADLOCK DETECTION ALGORITHM

A. DEFINITIONS

Before we proceed with the development of our deadlock detection algorithm, we need to more formally define what is meant by deadlock and related phenomena of interest. These phenomena are first defined for a general case involving a software process. Following an introduction to the Spec formal specification language, these phenomena are more precisely defined in terms of the Spec atomic transactions evaluated by our algorithm.

1. Deadlock

If we model a process as a directed graph $G = (V,E)$, where V is a set of process states, including a subset F of acceptable final states, and E is a set of transitions between states, a process has deadlocked if it is in a state V' not in F and there are no transitions from V' to any other state.

2. Starvation

Using the above model, a process is subject to starvation or permanent blocking, if it is in a state V', there exists an infinite path of transitions from V', and none of the states on the infinite path is in F. There may be a different finite path of transitions from V' to a state V'' in F.

3. Livelock

A process is in a state of livelock, if it is in a state V' from which there exists an infinite path of transitions in G, and there does not exist a path from V' to a state V'' in F. Livelock implies starvation, but not the converse.

4. Looping

Looping occurs when execution of a set of transitions in a process induces the execution of at least one infinite path of transitions even though some finite segment of the path may reach a state V' in F.

B. REQUIREMENTS FOR A DEADLOCK DETECTION TOOL

1. Approach

The first step in developing the deadlock detection tool is to establish the top level requirements for what the tool should accomplish. The next step is the selection of a particular specification methodology available for use in developing the underlying algorithm and testing the tool. The key elements of the specification method critical to deadlock detection then must be identified and a data structure allowing the use of these elements for the algorithm must created. Finally, the details of the algorithm must be determined.

2. Identification of End User

Certain user interface principles should be considered (Fisher, 1988). In particular, we identify the end user of this tool, and target the abilities and requirements of that user. The intended end user of this tool is a software engineer writing the specification for a distributed system. This end user understands the flow of data and messages between software modules. Integration of this tool into a computer aided design environment should increase its utility to the practicing software engineer.

We would like the tool to be able to process both a completed specification, and one that is partially developed, module by module. We would also like the tool to assist in the *elimination of deadlock from* the distributed system being evaluated in cases where a potential deadlock situation is detected.

3. Specification of Desired Input and Output of Tool

Given a formal specification of a distributed system as input, the tools must at a minimum determine the potential for deadlock in that distributed system.

A Boolean answer to the deadlock potential question is of limited utility. Information that assists the end user in compensating for, or eliminating identified deadlock potentials is also desired.

Thus, the tool should input either a partial or a complete specification, determine if there is a potential for deadlock, and if the potential for deadlock exists, the tool should help to determine cause or location of that potential deadlock. Extensions to this requirement could include tying a graphic representation to the text form so that a graphic editor could be used in eliminating the potential for deadlock. Full evaluation of user interface requirements are left for future work.

C. SELECTING THE SPECIFIC SPECIFICATION FORMALISM

We have selected Spec (Berzins, 1991), (Berzins and Luqi, 1990), (Berzins and Luqi, 1991) as the specification language for this tool. As described in Chapter II, there are many formal software specification methods available for use in specifying distributed systems. Included among these are temporal logic, transition axioms (Wing, 1990), Communicating Sequential Processes (CSP) (Hoare, 1985), Petri nets (Murata, 1989), Communicating Finite State Machines (Brand, 1983), Systems of Communicating Machines (Lundy, 1988), SDYMOL (Dillon, 1988), and Spec.

The use of constrained expressions for analyzing distributed system behavior has been evaluated for CSP, Petri nets and SDYMOL. The constrained expression approach can be used to provide a formal basis for arguments in analyzing the potential for deadlock in a distributed system (Dillon, 1988).

Spec also lends itself for evaluation using a constrained expression approach, although it may not be possible to express all Spec transactions using constrained expressions.

The specification methodology underlying our deadlock detection approach should support automated processing and CASE tool development and should have facilities for specifying parallel and distributed systems. This requirement rules out most of the specification methods discussed in Chapter II that satisfy the previous requirement. The specification language Spec satisfies both requirements. Spec has been chosen because of this and because it has been specifically designed for constructing large-scale software systems.

D. INTRODUCTION TO SPEC

The basic units or modules of Spec are definitions, functions, types, machines, and instances. Of these, only machines and types can be associated with atomic transactions that can lead to deadlock. A machine is a system with a memory capable of remembering a state. A type is a module that defines an abstract data type, consisting of a set of values and a set of primitive operations on that value set.

1. Atomic Transactions

Atomic transactions specify the synchronization requirements between potentially concurrent processes in a distributed environment. Spec *transactions* constrain the order in which *actions* consisting of *events* and *subtransactions* can occur within *machine* or *type* modules in a Spec specification.

Events correspond to arrivals of *messages* passed between modules. Events can trigger outgoing messages and can modify the internal state of a module. Events are instantaneous occurrences that trigger software responses and serve as time references for real-time constraints. Atomic transactions can require some events to be delayed. Thus messages must be buffered, and events occur when the software starts to process a message, rather than when the message physically arrives at a processor.

Actions in a transaction may represent nested subtransactions. The syntax of a Spec atomic transaction is shown in Figure 3.1. The instance declaration consists of optional WHERE and FOREACH clauses. The WHERE clause is used to specify timing constraints. FOREACH is used to describe a set of messages or instances. The action list for an atomic transaction can express sequencing of actions (one instance of each action listed must occur in the order listed, indicated by "action1; action2; action3..."), parallel combination of actions (one instance of each action listed must occur, in some unspecified order, indicated by "action1 action2 action3..."), choice of exactly one of the alternatives listed (delimited by the key words IF and FI, "IF alternative1 | alternative2 | alternative3 FI"), repeated choice (delimited

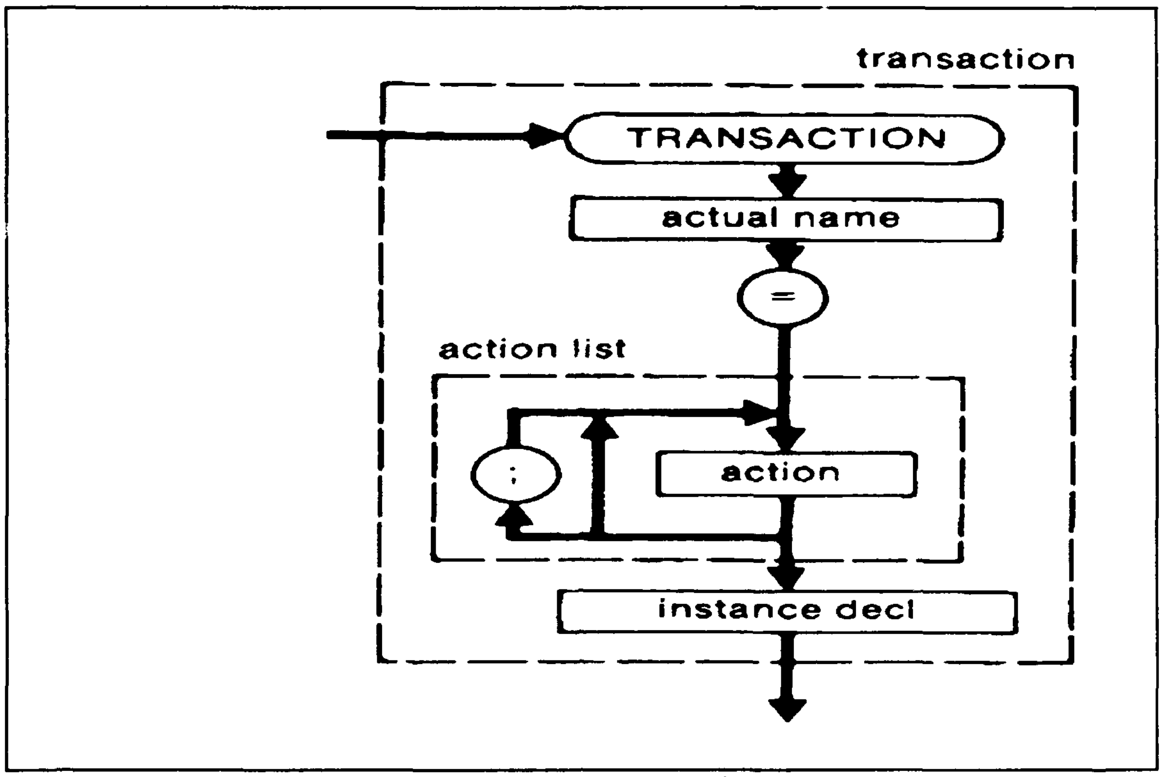

Figure 3.1 Spec Transaction Syntax Diagram

by the keywords DO and OD, indicating zero or more instances of the alternatives listed, "DO alternative1 | alternative2 | alternative3 OD"), or a combination of the above. Alternatives consist of an optional 'guard' (WHEN expression_list ->) followed by a list of actions to be performed. The guard conditions can depend on state variables of a machine, but not on data arriving in messages. The actions listed are either message names or names of other transactions.

The Spec statement:

```
TRANSACTION example =
            message1;
            IF message2 | WHEN A > B -> message3 FI;
            message4 message5;
            DO message6 OD;
            DO message7 | message8 OD;
            message9
```

indicates that the atomic transaction named "example" consists of the arrival of message1, followed by the arrival of either message2 or message3 (with a guard on message3 indicating that it may only be accepted if the value of expression "A" is greater than the value of expression "B"), followed by the arrival of message4 and message5, but in unspecified order, followed by zero or more instances of message6, then zero or more instances of either message7 and/or message8, and finishing with the arrival of message9.

We now refine our previous definitions for deadlock and other undesirable distributed system phenomena in terms of the structure of a Spec atomic transaction.

2. Deadlock

In a Spec transaction, we say that deadlock exists if there exists an execution path that terminates (reaches a "dead end" where no more events are possible) prior to completion of the atomic transaction. More formally, given a

Spec atomic transaction *T*, consisting of a set of actions, *S*, (including embedded atomic transactions), transaction *T* has the potential for deadlock if an only if there exists a path of execution through *S* that terminates in an action that does not complete *T* and that cannot be extended by any pending actions.

3. **Starvation**

Given a Spec atomic transaction, *T*, consisting of a set of actions, *S*, starvation may occur if and only if there exists an infinite path of execution through *S*, that is consistent with the order imposed on *S* in *T* and does not lead to completion of *T*. There may also exist another path of execution through *S* that completes *T*.

4. **Livelock**

Given an atomic transaction, *T*, consisting of a set of actions, *S*, livelock occurs if there exists an infinite path of execution through *S*, as constrained by the order imposed on *S* in *T*, and there does not exist a finite path of execution through S that completes the transaction *T*.

5. **Looping**

Given a Spec atomic transaction, *T*, consisting of a set of actions, *S*, looping occurs if execution of a path through *S* as constrained by the order specified in *T*, induces an infinite path of execution though *S*, even if some finite initial segment of the path may complete the transaction *T*.

E. DETECTING DEADLOCK IN SPEC SPECIFICATIONS

Evaluating Spec specifications for deadlock potential requires consideration of several key elements. These elements are not necessarily dependent on, or connected to one another.

1. Atomic Transactions Necessary for Deadlock to Occur

If there are no atomic transactions, deadlock cannot occur, since in such a case no module must wait for actions of any other module before it will accept another stimulus. Therefore we need check only those modules that participate in atomic transactions, but no others. Infinite loops within methods for individual messages are not an issue here because all of the specified responses to a stimulus are always required to appear in a finite time. Correct implementations of a Spec module may not get stuck in infinite loops.

2. Attributes for Deadlock Detection

Certain attributes of Spec atomic transactions are needed to evaluate those transactions for potential deadlock.

a. Unique Identification

Actions in a transaction must be uniquely identified. This can be done by using the name of the Spec *machine* or *type* module in which the atomic transaction is specified, and the name of an *actions* in the atomic transaction.

b. Message Preconditions

Associated with each message in the response to an event is a *precondition* that describes the conditions under which the response must be produced. The precondition is a logical assertion contained in the MESSAGE specification for the event.

c. Completion

The last property is whether or not a given sequence of actions in an atomic transaction completes a given transaction. This is analogous to a "final" or "accepting" state in a finite state machine.

3. Regular Expressions

The actions and triggering conditions of a transaction are determined from the message definitions in a Spec machine or type. If guard conditions are excluded from transactions and recursive subtransactions are not allowed, then the constraints imposed by an atomic transaction can be represented by regular expressions. This is the subset of synchronization constraints considered in this thesis.

4. Context Free Expressions

A more general case of Spec specifications noted above includes recursive subtransactions in atomic transactions. In this case the set of action sequences completing a transaction from a context free language.

The completion property introduced in 2.c. above is desired by constraining the cause/effect relationships contained in the message definitions by the definitions of all applicable atomic transactions. These constraints may be expressed by a context free grammar.

F. INTRODUCING THE ALGORITHM

1. Message Graphs

The cause/effect relationships between events defined by the specification can be modeled by a directed graph. Figure 3.2 depicts this representation. The actions of a transaction are the nodes of the graph (top), and the preconditions for those actions are used to label the edges of the graph (bottom). Representing the behavioral specification of a message in this model yields a *message graph*.

Using this model, we derive a directed graph from the Spec specification for a distributed system. We analyze the definition for each *message* used in a transaction, creating a *message graph* representing the possible response paths specified for that message. The algorithm to create a *message graph* is shown in Figure 3.3.

The message graph for each message consists of a root labeled with the message name and module in which defined, a set of response edges representing the possible paths from the root node, each labeled with the precondition for taking that path, and a set of leaf nodes representing the response set of

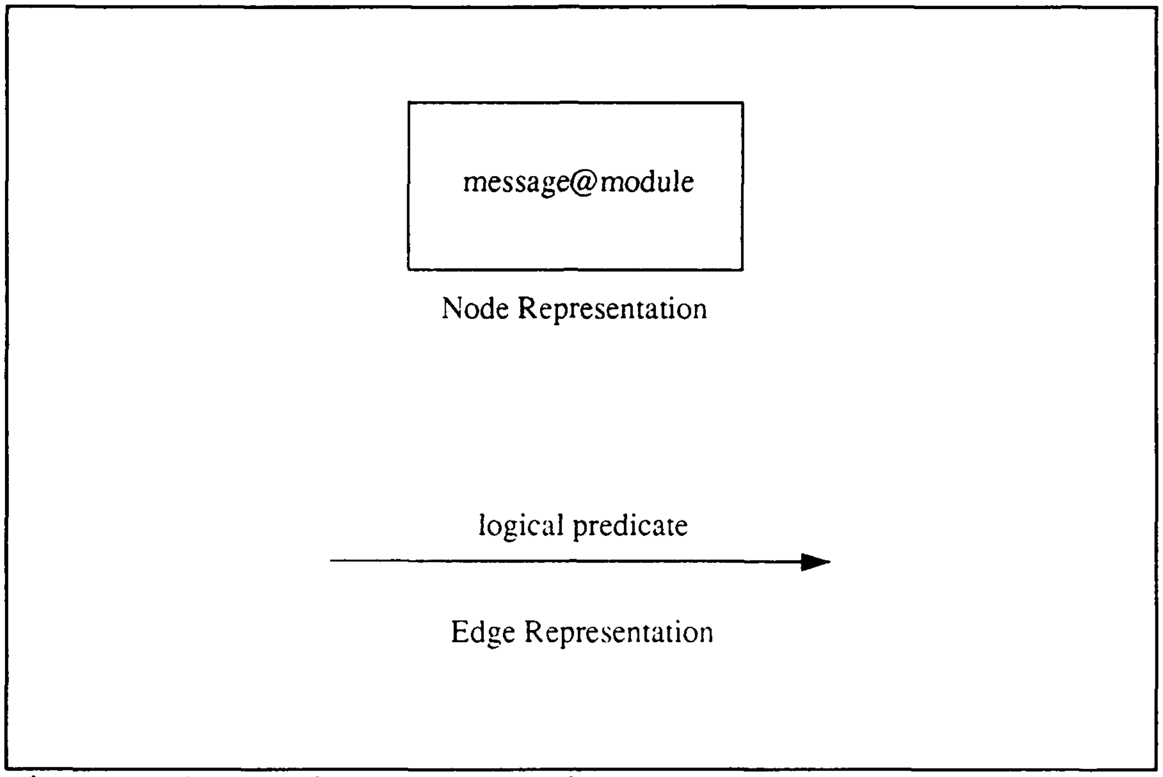

Figure 3.2 Graph Representations

actions resulting from the preconditions represented by the response edges. The message preconditions must be checked to ensure that all possible values for the module state and message data variables are considered. If not, the message definition is incomplete. The message graph for the following Spec excerpt is depicted in Figure 3.4:

```
        MESSAGE is_positive (i : integer)
           WHEN i > 0
               SEND positive TO monitor
           OTHERWISE SEND not_positive TO monitor
```

```
Algorithm Create_message_graph(a : message, M : module,
                                MG : message_graph) is

Input: M = actual_name of module where the message a
                is defined

Output: MG = (V, E) (a message_graph) where
                V : node_list (initially empty);
                E : edge_list (initially empty);

begin

        add node a@M to V;

        for each response case in a
          for each response in the response case
            define b = name of actual message in the
                          reply or send
            define N = name of caller for a reply or
                          actual_name of recipient for
                          a send

            if node b@N is not in V then
                add b@N to V;
            end if;

            if response case has a guard condition
                edge_label is guard expression_list;
            else
                edge_label is true;
            end if;

            add edge from a@M to b@N labeled with
                edge_label to E;
          end for;
        end for;

end Create_message_graph;
```

Figure 3.3 Algorithm Create_message_graph

The entire specification is parsed, module by module, and Create_message_graph is used to create a data base of message graphs representing all messages in the specification.

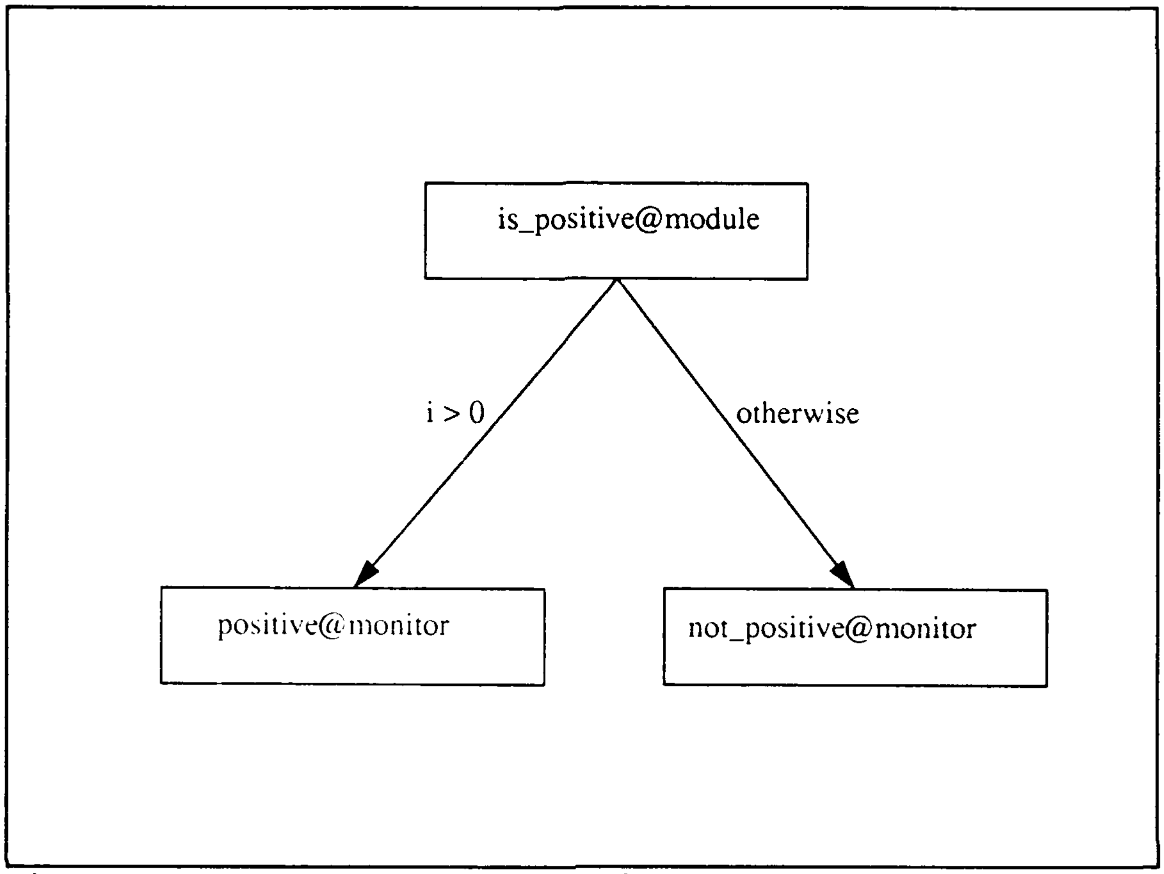

Figure 3.4 Example Message Graph

2. Partial Behavior Graphs

After the message graphs are created, the transaction being evaluated is scanned to determine the *alphabet* of the transaction, that is, which messages are used by the transaction. In the event that an action in a transaction is a subtransaction instead of a message the algorithm for determining the alphabet of a transaction is called recursively. The message graphs for this alphabet are then merged into a *behavior graph* for the transaction.

The message graphs identified in the alphabet of the transaction comprise only the initial portion of the behavior graph. Figure 3.5 shows the specification of the sender module in an example sender-receiver protocol described by (Berzins and Luqi, 1991). The partial behavior graph for transaction *transfer* in the sender protocol of Figure 3.5 is shown in Figure 3.6.

3. Behavior Graphs

Once the message graphs identified by the transaction alphabet are added to the behavior graph, the message graphs represented by leaf nodes (nodes with out-degree zero) of the partial behavior graph are repeatedly added, until the graph does not change any more. Figure 3.7 depicts the completed behavior graph for transaction *transfer@sender*. The dashed lines in the figure denote edges defined in message graphs outside *MACHINE sender*. Assuming that the atomic transaction includes only actions previously defined (if not, the specification is incomplete), this behavior graph represents all possible execution paths for the messages included in the transaction. The graph does not reflect the constraints on order of execution specified in the definition of the transaction.

The specification of the accompanying receiver module for our example is shown in Figure 3.8. The partial and complete behavior graphs for transaction *receive@receiver* are

```
MACHINE sender
   STATE(data: sequence{block}
   INVARIANT true
   INITIALLY data = [ ]

   MESSAGE send(file:  sequence{block})
     WHEN length(file) > 0
       SEND first(b: block) TO receiver WHERE b = file[1]
       TRANSITION data = file
     OTHERWISE REPLY EXCEPTION empty_file

   MESSAGE echo(b: block)
     WHEN b = data[1] & length(data) > 1
       SEND next(b1: block) TO receiver WHERE b1 = data[1]
       TRANSITION *data = b || data
     WHEN b = data[1] & length(data) = 1
       SEND done TO receiver
       SEND done TO sender
       TRANSITION data = [ ]
     OTHERWISE SEND retransmit(b2: block)
       TO receiver WHERE b2 = data[1]

   MESSAGE done
   TRANSACTION transfer = send; DO echo OD ; done
END
```

Figure 3.5 Specification of a Sender Protocol

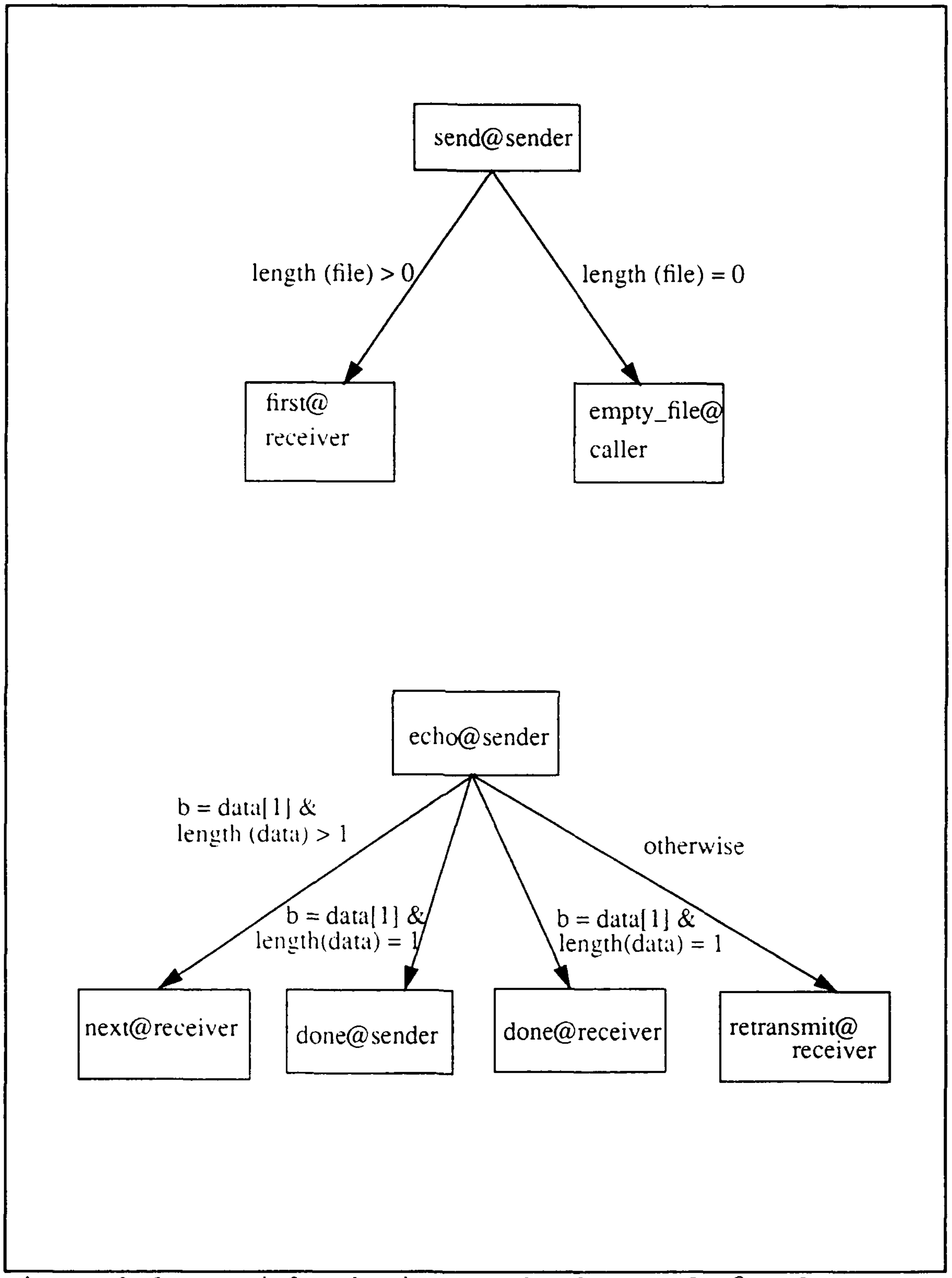

Figure 3.6 Partial Behavior Graph of *transfer@sender*

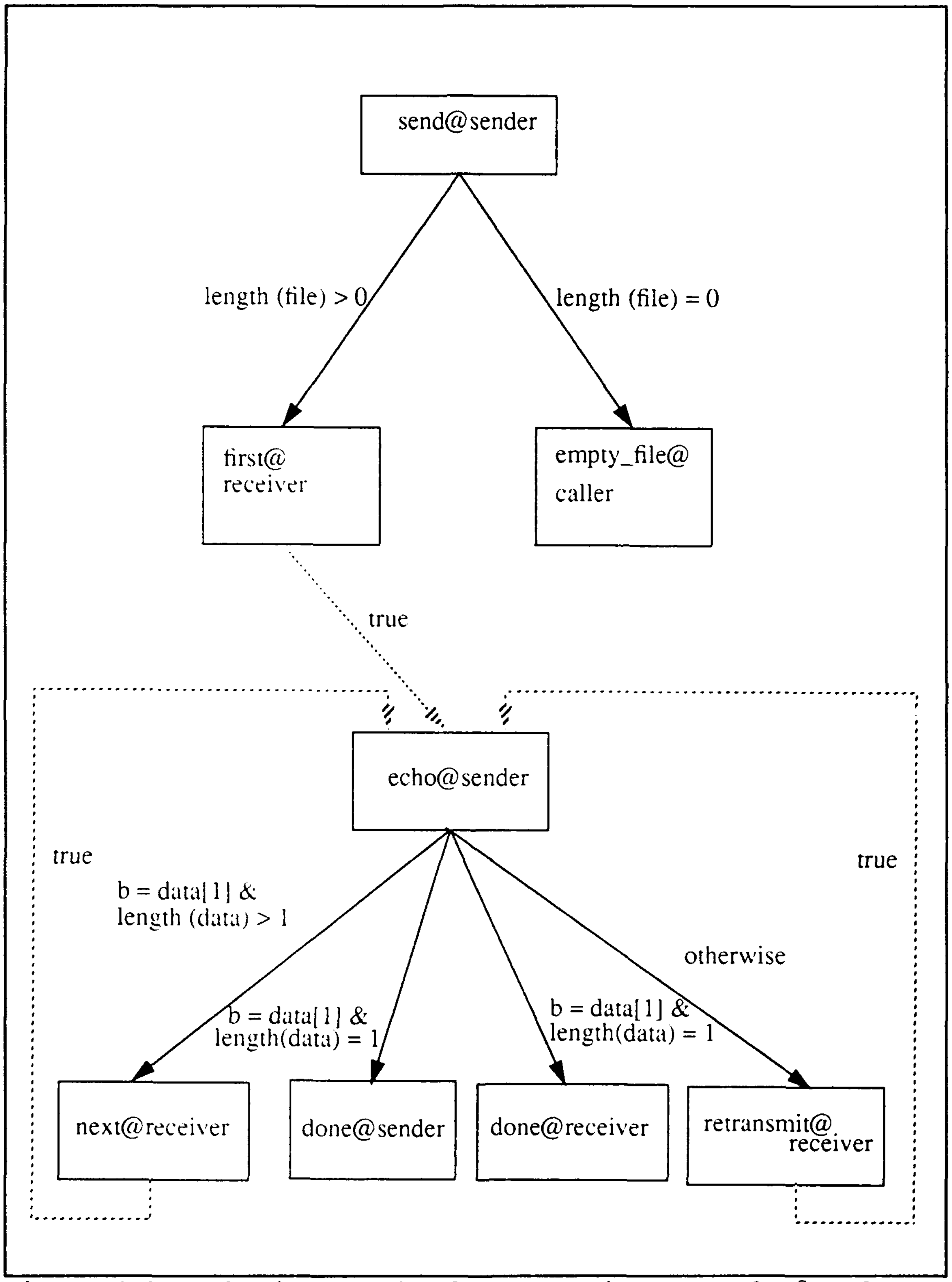

Figure 3.7 Behavior Graph of Transaction *transfer@sender*

```
MACHINE receiver
   STATE(data: sequence{block}
   INVARIANT true
   INITIALLY data = [ ]
   MESSAGE first(b: block)
     SEND echo(b: block) TO sender
       TRANSITION data = [b]
   MESSAGE next(b: block)
     SEND echo(b: block) TO sender
                TRANSITION *data = data || b

   MESSAGE retransmit(b: block)
     SEND echo(b: block) TO sender
                TRANSITION data[length(*data)] = b
         -- Replace the last element of data.

   MESSAGE done
     SEND save(file: sequence{block})
       TO file_system WHERE file = data
     TRANSITION data = [ ]

   TRANSACTION receive = first; DO next | retransmit OD ; done

END
```

Figure 3.8 Specification of a Receiver Protocol

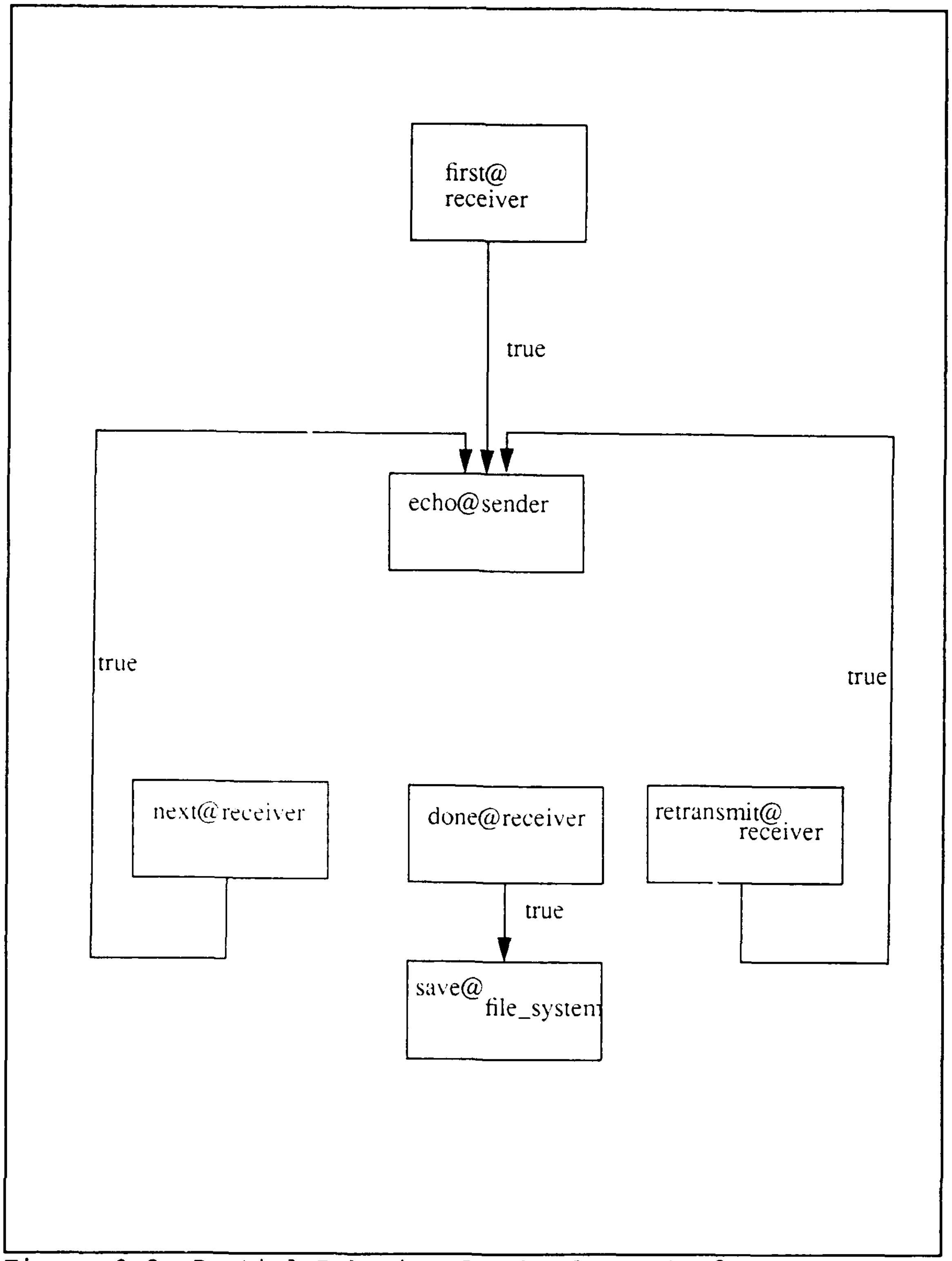

Figure 3.9 Partial Behavior Graph of *receive@receiver*

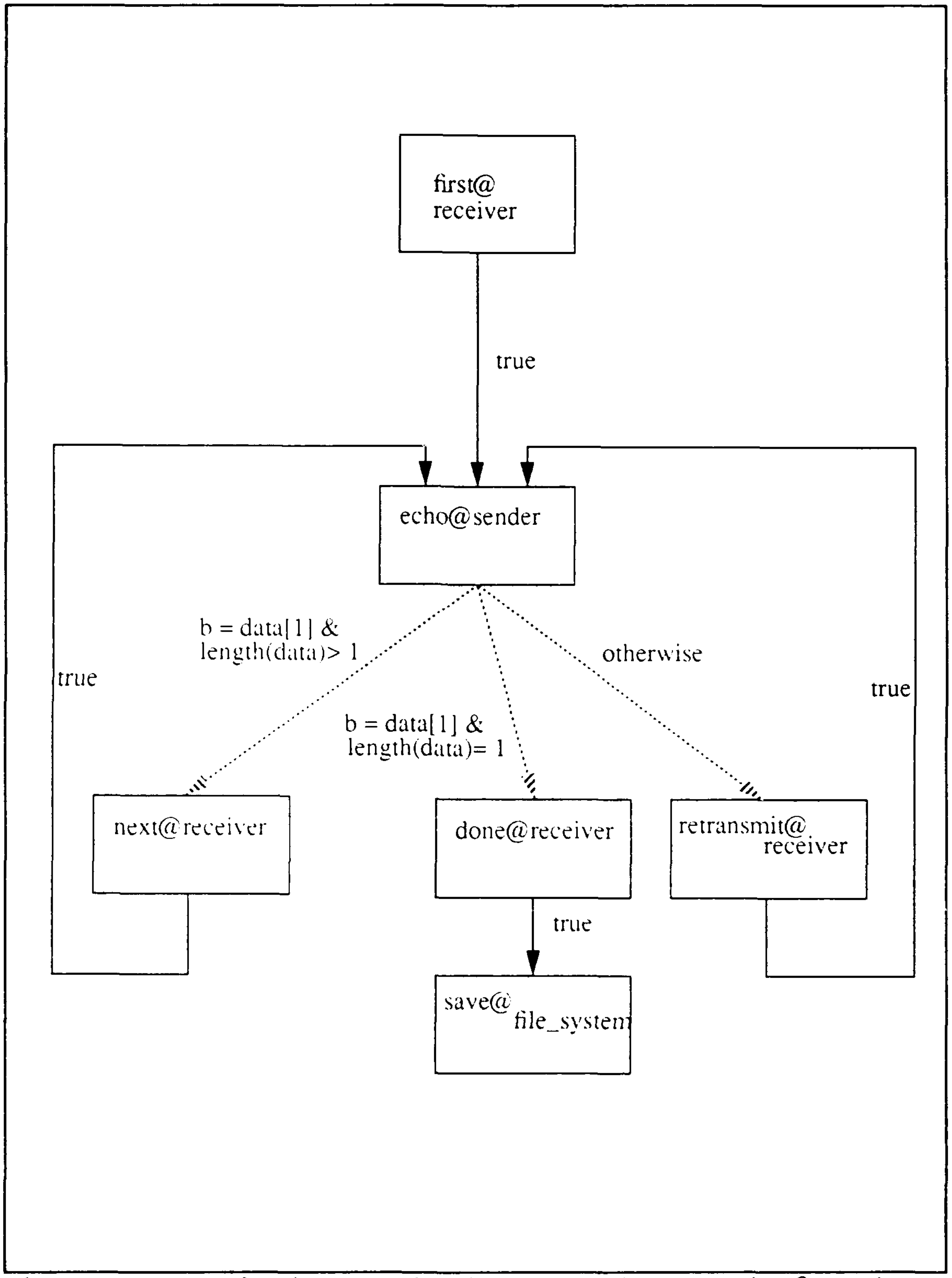

Figure 3.10 Behavior Graph of Transaction *receive@receiver*

shown in Figures 3.9 and 3.10. The behavior graphs are similar to, but not identical with the stimulus-response graph described by (Berzins and Luqi, 1991) (Figure 3.11 in our example).

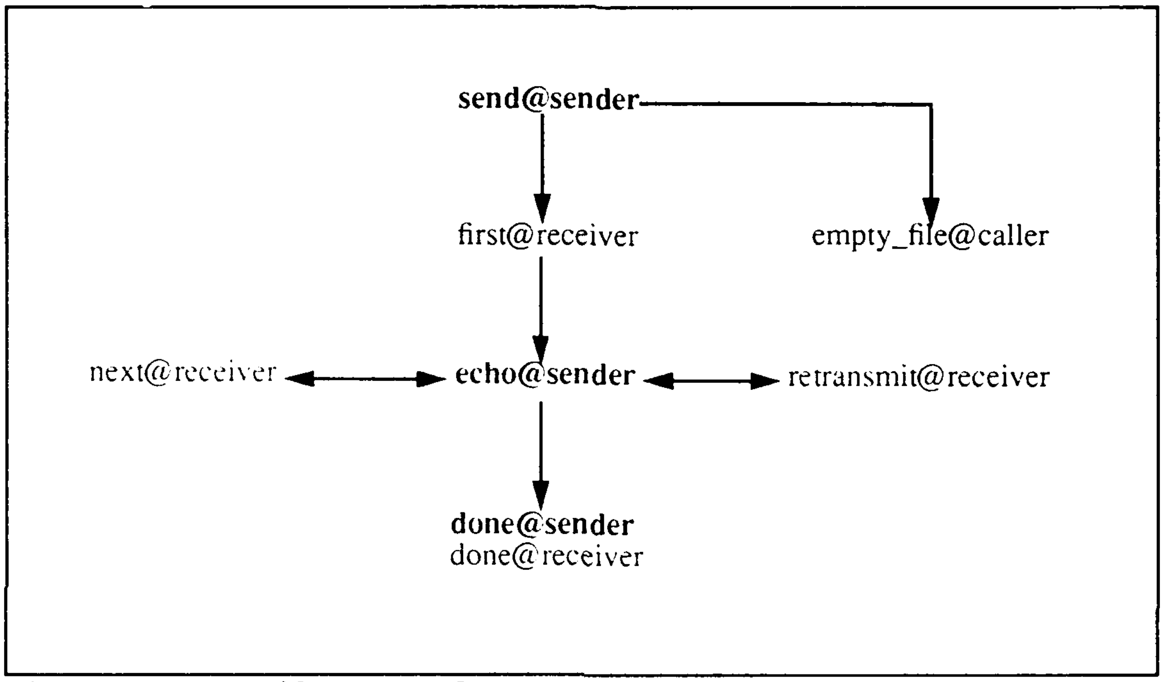

Figure 3.11 File Transfer Protocol Stimulus Response Graph

4. Reduced Behavior Graphs

Further analysis can be simplified by first removing nodes not in the alphabet of the transaction from the behavior graph. We must ensure that we do not either create or delete possible paths of execution during the process of creating the *reduced behavior graph*. Therefore, there must be a one-to-one correspondence between restrictions of the paths in the behavior graph to the alphabet of the transaction and the paths in the reduced behavior graph. The behavior graph

```
Algorithm Reduce_behavior_graph(BG : behavior_graph,
                             T : transaction alphabet,
                             RBG : reduced_behavior_graph) is

Input: BG = (V, E) (a behavior_graph with
                             V : node_list;
                             E : edge_list;
       and T = the alphabet of the transaction used to build
               BG

Output: RBG = (V', E') (a reduced_behavior_graph) where
        V' : node_list (initially set to V);
        E' : edge_list (initially set to E);

begin

   for each node n in BG but not in T

      for each incoming edge $e_i$ to n

         for each outgoing edge $e_o$ from n

            add edge from source($e_i$) to destination($e_o$) to E';
            remove $e_o$ from E';

         end for;

         remove $e_i$ from E';

      end for;

      remove n from V';

   end for;

end Reduce_behavior_graph;
```

Figure 3.12 Algorithm Reduce_Behavior_Graph

reduction algorithm is shown in Figure 3.12. Figure 3.13 depicts this *reduced behavior graph* for transaction *receive@receiver*. Analysis for safety properties other than deadlock (starvation, livelock, looping) may require using the full behavior graph.

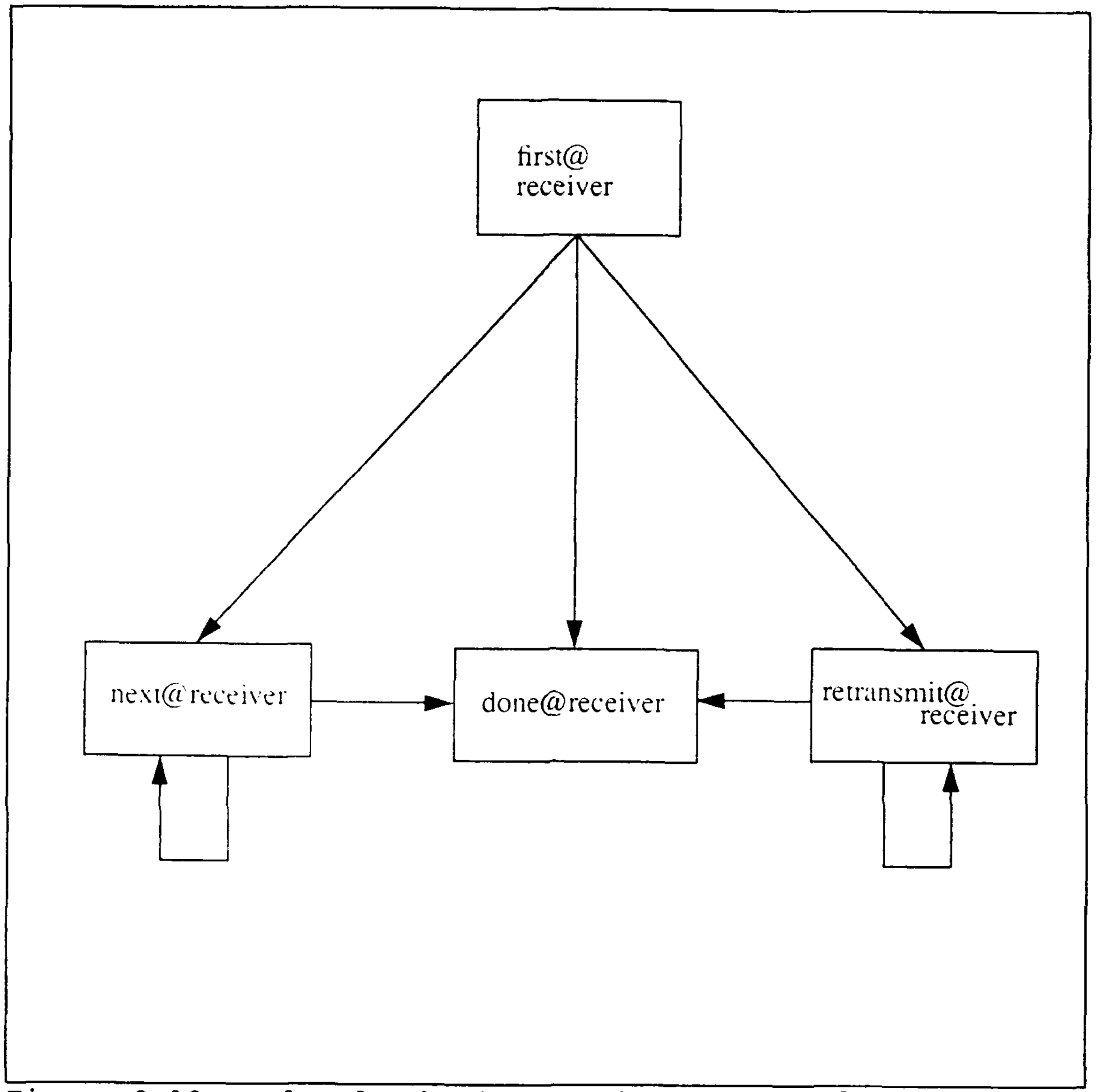

Figure 3.13 Reduced Behavior Graph of receive@receiver

5. Constraining the Behavior Graph

The behavior graph (reduced, if appropriate) of the transaction is then constrained by the order of execution specified in the definition of the transaction. A reachability analysis is performed using the possible execution paths in the behavior graph. Execution paths terminate at leaf nodes of the behavior graphs. Any potential deadlock will be represented by a leaf node being encountered prior to one of the completion messages of the transaction being accepted.

If a message from an atomic transaction is sent to a module whose specification is not yet complete, a potential deadlock might be indicated, even though one might not actually occur. This is due to the incomplete specification.

G. COMPLEXITY

While it is indeed possible for execution paths through the transactions being evaluated to infinite, the number of actions and paths in the behavior graph is finite (since we are dealing with regular expressions). Reachability analysis of the type used here has been previously shown to decidable, though requiring exponential time and space in the worst case (Kosaraju, 1982, Mayr, 1984).

IV. CONCLUSIONS

A. SUMMARY

In this paper, I have discussed issues in developing a software tool for detecting potential synchronization constraint deadlock in distributed systems from analysis of the formal specifications of those system. I reviewed several formal software specification methods and indicated why Spec is the appropriate specification language for this work.

I established requirements for the functionality of the deadlock detection tool, and identified necessary conditions for and attributes of deadlock occurrence and detection.

I described a potential method based on graph theory for deadlock detection analysis for specifications whose synchronization constraints can be represented using regular expressions. This algorithm was illustrated via an example, and its complexity discussed.

B. IMPORTANCE OF RESEARCH RESULTS

This work supports the development of a software tool for detecting potential deadlock from the formal specification of a distributed system. None of the software tools presently available have demonstrated this capability. While the type of deadlock evaluated represents a subset of those that might occur, and with specifications that can be analyzed representing a subset of those that can be described using Spec, this research is additionally important because it uses as its base a specification language that both can be directly used for specifying large-scale concurrent systems, and is intended for use by system designers.

C. PROPOSED EXTENSIONS

Future work includes a more rigorous description of the algorithm and its analysis for correctness and complexity, and actual implementation of the tool. Research into extending the algorithm to specifications not restricted to a regular grammar representation is also necessary.

In conjunction with the deadlock detection work already noted, additional research includes the development of a methodology for modifying the specification to preclude the potential for deadlock, once a potential for deadlock is determined.

LIST OF REFERENCES

Ambler, A. L., "GYPSY: A Language for Specification and Implementation of Verifiable Programs," *Proceedings of an ACM Conference on Language Design for Reliable Software, ACM Software Engineering Notes*, Vol. 2, No. 2, pp. 1-10, March 1977.

Beizer, B., *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, 1990.

Berzins, V., and Luqi, *Languages for Specification, Design, and Prototyping,* Naval Postgraduate School NPS52-88-038, September 1988.

Berzins, V., and Luqi, "An Introduction to the Specification Language Spec," *IEEE Software*, pp. 74-84, March, 1990.

Berzins, V., and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, 1991.

Berzins, V., "Black-Box Specification in Spec," *Computer Language*, Vol. 16, No. 2, pp. 113-127, 1991.

Bochmann, G. V., "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers*, Vol. C-31, No., 3, pp. 223-231, March 1982.

Brand, D. and Zafiropulo, P., "On Communicating Finite State Machines," *Journal of the ACM*, Vol. 30, No. 2, pp. 323-342, Apr 1983.

Brooks, S. D., and Roscoe, A. W., *Deadlock Analysis in Networks of Communicating Processes*, Carnegie Mellon University CMU-CS-89-161, June 1989.

Carranza, M., and Young, W. D., *An Annotated GYPSY Bibliography*, Computational Logic, Inc. Internal Note 105, August 1989.

Chandy, K. N., Misra, J., and Haas, L, M., "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144-156, May 1983.

Cohen, B., "A Rejustification of Formal Notations," *Software Engineering Journal*, Vol. 4, No. 1, pp. 36-38, January 1989.

Cooke, D. E., "Formal Specifications of Resource-Deadlock Prone Petri Nets," *Journal of Systems and Software*, Vol. 14, No. 11, pp. 53-69, November 1990.

Coolahan, J. E., and Roussopoulos, N., "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp. 603-616, September 1983.

Deitel, H. M., *An Introduction to Operating Systems, Second Edition*, Addison-Wesley, 1990.

Dillon, L. K., Avrunin, G. S., and Wiledon, J. C., "Constrained Expressions: Toward Broad Applicability of Analysis Methods for Distributed Software Systems", *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 3, pp. 374-402, July 1988.

Dillon, L. K., "Verifying General Safety Properties of Ada Tasking Programs," *IEEE Transactions on Software Engineering*, Vol. 16, No. 1, pp. 51-63, January 1990.

Fisher, A. S., *CASE: Using Software Development Tools*, John Wiley & Sons, 1988.

Goltz, U., "CCS and Petri Nets," *Semantics of Systems of Concurrent Processes*, Lecture Notes in Computer Science 469, pp. 334-357, Springer-Verlag, 1990.

Good, D. I., DiVito, B. L, and Smith, M. K., *Using the Gypsy Methodology*, Computational Logic, Inc. Technical Report CLI-2, January 1988.

Gouda, M. G., and Chang, C., "Proving Liveness for Networks of Communicating Finite State Machines," *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 1, pp. 154-182, January 1986.

Guttag, J. V., Horning, J. J., and Wing, J. M., "The Larch Family of Specification Languages," *IEEE Software*, pp. 24-36, September, 1985.

Harel, D., and others, "STATEMENT: A Working Environment for the development of Complex reactive Systems," *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 4, pp. 403-414, April 1990.

Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

Hopcroft, J. E., and Ullman, J. D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

Jones, C. B., *Systematic Software Development using VDM*, Second Edition, Prentice Hall, 1990.

Kosaraju, S. R., "Decidability of Reachability in Vector Addition Systems, "*Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pp. 267-281, 1982.

Luckham, D. C., and von Henke, F. W., "An Overview of Anna, a Specification Language for Ada," *IEEE Software*, pp. 9-22, March, 1985.

Luckham, D., *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*, Texts and Monographs in Computer Science XVI, Springer-Verlag, 1990.

Lundy, G. M., *Systems of Communicating Machines: A Model for Communication Protocols*, Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1988.

Masticola, S. P., and Ryder, B. G., "A Model of Ada Programs for Status Deadlock Detection in Polynomial Time," *ACM SIGPLAN Notices*, Vol. 26, No. 12, pp. 97-107, December 1991.

Mayr, E. W., "An Algorithm for the General Petri Net Reachability Problem," *SIAM Journal on Computing*, Vol. 13, No. 3, pp. 441-460, August, 1984.

Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

Morgan, E. T., *RGA Users Manual*, Version 2.3, University of California, Irvine, Department of Information and Computer Science Technical Report 87-04, January 1987.

Murata, T., Shenker, B., and Shatz, S. M., "Detection of Ada Static Deadlocks Using Petri Net Invariants," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, pp. 314-326, March 1989.

Murata, T., "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, April, 1989.

Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 455-495, July 1982.

Peterson, J. L., "Petri Nets," *Computing Surveys*, Vol. 9, No. 3, pp. 223-252, September 1977.

Pnueli, A., "The Temporal Semantics of Concurrent Programs," *Semantics of Concurrent Computation*, Lecture Notes in Computer Science 70, pp. 1-20, Springer-Verlag, 1979.

Spivey, J. M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

Spivey, J. M., "An Introduction to Z and Formal Specifications," *Software Engineering Journal*, Vol. 4, No. 1, pp. 40-50, January 1989.

Sudkamp, T. A., *Languages and Machines, An Introduction to the Theory of Computer Science*, Addison-Wesley, 1988.

Tanenbaum, A. S., *Structured Computer Organization, Second Edition*, Prentice-Hall, 1984.

Taylor, R. N., "A General-Purpose Algorithm for Analyzing Concurrent Programs," Communications of the ACM, Vol. 26, No. 5, pp. 362-376, May 1983.

Valenzano, A., Sisto, R., and Ciminiera, L., "An Abstract Execution Model for Basic LOTOS," *Software Engineering Journal*, Vol. 5, No. 6, pp. 311-318, November 1990.

Wing, J. M., "An Specifier's Introduction to Formal Methods", *Computer*, Vol. 23, No. 9, September, 1990.

Woodcock, J. C. P., and Morgan, C., "Refinement of State-Based Concurrent Systems," *VDM '90: VDM and Z - Formal Methods in Software Development*, Lecture Notes in Computer Science 428, pp. 340-351, Springer-Verlag, 1990.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Prof. Valdis Berzins Code CS/Be Naval Postgraduate School Monterey, CA 93943	2
4.	Prof. Luqi Code CS/Lq Naval Postgraduate School Monterey, CA 93943	1
5.	Prof. Robert McGhee Code CS/Mz Naval Postgraduate School Monterey, CA 93943	1
6.	Prof. Man-Tak Shing Code CS/Sh Naval Postgraduate School Monterey, CA 93943	1
7.	Prof. G. M. Lundy Code CS/Ln Naval Postgraduate School Monterey, CA 93943	1
8.	Prof. Carl R. Jones Code CC Naval Postgraduate School Monterey, CA 93943	1
9.	Prof. James N. Eagle Code AW/Er Naval Postgraduate School Monterey, CA 93943	1

10.	Dr. R. Wachter (Code 1133) Computer Science Division Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
11.	Jeffrey M. Schweiger 44 Roundabout Road Smithtown, NY 11787-1841	1
12.	LCDR J. M. Schweiger, USN Patrol Wing ONE Detachment Diego Garcia FPO AP 96464-0021	1